Parallelizing Wavelet Transformation with a Reduction Style Framework

Venkatram Ramanathan  Vignesh T. Ravi  Gagan Agrawal
Department of Computer Science and Engineering
The Ohio State University, Columbus, OH 43210
{ramanath,raviv,agrawal}@cse.ohio-state.edu

ABSTRACT
There is an increasing trend towards parallel processing and data processing using cloud environments. Simple APIs for expressing parallelism, such as map-reduce, are popular in cloud environments, but their expressibility is generally considered limited.

This paper focuses on how a signal processing algorithm, wavelet transform, can be parallelized in a cloud environment comprising a cluster of multi-cores using a reduction-based framework. Particularly, we use a framework, FREERIDE, which has many similarities with map-reduce, but its API involves an explicit reduction object. We show how all communication in this algorithm can be managed using a reduction object. Besides programming with a simpler model, we also achieve good parallel efficiency, with a speedup of up to 42 on 64 cores.

1. INTRODUCTION
The availability of large datasets and increasing importance of data analysis in commercial and scientific domains is creating a new class of high-end applications. Recently, the term Data-Intensive SuperComputing (DISC) has been gaining popularity [2], reflecting a growing class of applications that perform large-scale computations over massive datasets. The deluge of available data for analysis demands the need to scale the performance of data mining implementations.

The emerging cloud environments are well suited for storage and analysis of large datasets, since they can allow on-demand access to resources. However, developing high-performance implementations of data analysis tasks is a challenging problem. All HPC environments today, including clouds, typically comprise a cluster of multi-core machines. These systems have multi-level parallelism which can be hard to program or obtain high parallel efficiency from. One approach for parallelizing applications on a cluster of multi-core machines is to use a hybrid programming model, i.e., use MPI for distributed memory programming, and either OpenMP or Pthreads for shared memory parallelism within a node. This, however, can require a high-level of effort and expertise for scaling a single application.

Reduction-based frameworks have become popular [4, 9] and are well suited for programming in Clouds, because parallelism is expressed in an architecture-independent fashion. However, the class of applications that can be expressed using these frameworks is considered to be limited. In this paper, we focus on an application based on discrete wavelet transformation algorithm [13]. Wavelet transformation is an important tool in signal processing applications such as data and image compression, noise reduction, pattern recognition and fMRI processing. The algorithm itself requires several shifts of data, and indeed, the previous parallel algorithms for this problem have been communication-intensive [3, 11, 12].

These parallel algorithms do not appear suitable for mapping to a reduction-based framework.

We develop a new approach for parallelizing discrete wavelet transformation, which makes it suitable for implementation on a reduction-based framework. We implement this framework using a middleware called FREERIDE (FRamework for Rapid Implementation of Data-mining Engines) developed in our prior work [8, 9, 10]. The initial design of FREERIDE was based upon the observation that the structure of parallel algorithms for several popular data mining problems essentially involves a generalized reduction. This observation is used to support parallelization on both distributed memory and shared memory settings. FREERIDE offers a reduction based API, with many similarities to the map-reduce API [4]. However, our recent work has shown a large performance advantage with FREERIDE for several data mining applications [7].

Our work is particularly driven by medical imaging, and more specifically, functional MRI (fMRI). Medical imaging is an emerging and important area requiring large-scale data analysis. This area includes techniques and processes used to create and analyze images of human anatomy and physiology, with the goal either being clinical use for monitoring disease and treatment, or achieving advances in medicine. fMRI is fast becoming the pre-eminent imaging method for neuroimaging and is being used to explore and probe the mechanism of brain activation [6]. Functional neuroimaging scanners acquire spatio-temporal scalar data, with a timeseries associated at every 3D voxel. The 4D time-series neuro-
imaging is seldom viewed directly, but instead is subject to elaborate processing, analysis and inference [6, 1, 5]. The underlying algorithms essentially seek spatial-temporal patterns from noisy and poorly resolved datasets. Wavelet transform is often used as the method for noise reduction.

Since in fMRI and other similar applications, the behavior of the physical system is being studied across a time series, the wavelet transform is carried out across the time series on each of the points on the physical system. The dataset can thus be visualized as a hypercube with multiple cubes along the fourth dimension representing the time series as shown in Figure 1. Considering the fact that these datasets can be very large, parallelizing this application on a cluster of multi-cores will be useful.

This paper focuses on the use of FREERIDE for creating a parallel and scalable implementation of the wavelet transform algorithm applied on a four dimensional dataset, such as those obtained from fMRI. Our implementation is experimentally evaluated on a cluster of multi-cores using several different datasets. We show a speedup of up to 42 using 64 cores.

The rest of the paper is organized as follows. Section 2 describes the wavelet transform algorithm and discusses possible ways of parallelizing it. Section 3 describes the FREERIDE middleware which is used for our implementation. Section 4 describes how the wavelet transform algorithm can be expressed as a generalized reduction operation and discusses how the algorithm was parallelized on FREERIDE. Section 5 presents the experimental results. We conclude the paper in Section 6.

2. SEQUENTIAL WAVELET TRANSFORM

```c
index=0;
for(factor = NUM_TIME_SERIES/2-1; factor > 0; factor/=2) {
    for(t = 0 to factor) {
        output[index] = input[2*t] + input[2*t+1];
        index++;
    }
}
```

Figure 2: Sequential Discrete Wavelet Transformation Over One Time-Series

The Discrete Wavelet Transform is defined for a sequence of input that has 2^n numbers, representing a time-series of length 2^n. The output of the wavelet transform is a convolution along the time domain, resulting in a series again of 2^n values. The particular transform in consideration, the Haar wavelet transform, can be viewed as follows. We simply pair up input values, store their difference, and pass their sum. This process is repeated till we get 2^n - 1 differences and one final sum. The algorithm is formally shown in Figure 2. The first statement inside the inner loop stores the difference in the output array, and the second statement passes over their sum to the next iteration. To save space, the values are copied on to the input array itself. Each iteration of the outer loop involves half the steps of the previous iteration. In all, there are log(N) iterations of the outer loop.

To illustrate this algorithm, consider a wavelet transform over a time series of length 8 over the elements a_1, a_2, ..., a_8. The result is the following series: a_1 - a_2, a_3 - a_4, a_5 - a_6, a_7 - a_8, a_1 + a_2 - a_3 - a_4, a_5 + a_6 - a_7 - a_8, a_1 + a_2 + a_3 + a_4 - a_5 - a_6 - a_7 - a_8, and finally, a_1 + a_2 + ... + a_8.

Let the time series be of the length N = 2^n, and the elements be denoted as a_1, a_2, ..., a_N. It can be seen that an element at the index i in the input array contributes to the values at indices i/2, i/4 + N/2, i/8 + N/2 + N/4, ..., N - 2, and N - 1 (with output indices ranging from 0 to N - 1). Formally, it contributes to n values, whose indices can be summarized as

$$\sum_{j=2^{-k}}^{N} \frac{N}{2^{j-1}} + i$$

with k ranging from 1 to n, and the last value, at the index N - 1.

A straight-forward parallelization of this algorithm for a distributed memory setting can be very communication-intensive. This can be seen from the last two values of the output. The second last value in the wavelet transform of a given input is the difference between the sum of the first N/2 elements and the sum of the second N/2 elements. The last value of the wavelet transform is the sum of all the elements in the input series. This implies that if there are P nodes, last node, which will contain the last N/P values of the output, will require all the values from all the nodes. Even the first node can only calculate N/2P final values with the data that it has. In order to generate the remaining N/2P values, it requires the data from the second node. Besides the communication costs, another challenge is the programmability of such an application on a multi-core cluster. A hybrid programming model, i.e., combination of MPI and OpenMP, will require a high level of expertise and effort.

From programmability viewpoint, one way of parallelizing an application like this would be to use the map-reduce framework. The map-reduce programming model can be summarized as follows [4]. The computation takes a set of input points and produces a set of output {key, value} pairs. The user of the map-reduce library expresses the computation as two functions: Map and Reduce. Map, written by the user, takes an input point and produces a set of intermediate {key, value} pairs. The map-reduce library groups together all intermediate values associated with the same key and passes them to the Reduce function. The Reduce function, also written by the user, accepts a key and a set of values for that key. It merges together these values to form a possibly smaller set of values. Typically, only zero or one output value is produced per Reduce invocation.

In parallelizing the Wavelet Transformation using the map-reduce programming model, each element a_i can be read once and O(log(N)) {key, value} pairs can be generated. Here, key refers to the index of the output element, and value is either a_i or -a_i. Once the map operation is done, the N log(N) output pairs with N distinct values of key can be sorted. The reduce function will just be the summation of values.

In the given example, when a_1 is read, the following {key, value} pairs are output by the map function: {0, a_1}, {4, a_1}, {6, a_1}, and {7, a_1}. Similarly, when a_2 is read, the following {key, value} pairs are produced: {0, -a_2}, {4, a_2}, {6, a_2}, and {7, a_2}. This process continues for all the 8 values of input. The map-reduce library sorts and groups together all the intermediate values associated with the same key and passes them to the Reduce function. The reduce function will accumulate values belonging to each key by summation.

This method however, is inefficient for two main reasons. Firstly, as the size of the input time series increases, storing the O(N log(N)) intermediate output pairs becomes extremely expensive. For instance if there are 131072 points in the time series, the size of the intermediate {key, value} pairs will be 17 times the size of...
the input data itself. Second, sorting the intermediate values in a distributed environment will involve a high cost. Though some of this cost can be reduced by having a combine function within each node, the volume of intermediate data will still be unmanageable.

This paper focuses on the use of another reduction based framework, which can provide a similar programming API as map-reduce, but can help alleviate the above costs. We discuss this framework in the next section, and then describe our parallelization approach for wavelet transforms in Section 4.

3. FREERIDE MIDDLEWARE AND PARALLELIZATION TECHNIQUES

Now we describe the middleware FREERIDE, which is the basis for our work. This middleware system for cluster-based data-intensive processing shares many similarities with the map-reduce framework. However, there are some subtle but important differences in the API offered by these two systems. First, FREERIDE allows developers to explicitly declare a reduction object and perform updates to its elements directly, while in Hadoop/map-reduce, the reduction object is implicit and not exposed to the application programmer. Another important distinction is that, in Hadoop/map-reduce, all data elements are processed in the map step and the intermediate results are then combined in the reduce step, whereas in FREERIDE, both map and reduce steps are combined into a single step where each data element is processed and reduced before the next data element is processed. This choice of design avoids the overhead due to sorting, grouping, and shuffling, which can be significant costs in a map-reduce implementation. Furthermore, this also alleviates the need for storage of intermediate (key, value) pairs, which can require a large amount of memory, and slows down several data mining implementations [7].

The following functions must be written by an application developer as part of the API:

**Local Reductions:** The data instances owned by a processor and belonging to the subset specified are read. A local reduction function specifies how, after processing one data instance, a reduction object (declared by the programmer), is updated. The result of this process must be independent of the order in which data instances are processed on each processor. The order in which data instances are read from the disks is determined by the runtime system.

**Global Reductions:** The reduction objects on all processors are combined using a global reduction function.

**Iterator:** A parallel data-intensive application comprises of one or more distinct pairs of local and global reduction functions, which may be invoked in an iterative fashion. An iterator function specifies a loop which is initiated after the initial processing and invokes local and global reduction functions.

Throughout the execution of the application, the reduction object is maintained in main memory. After every iteration of processing all data instances, the results from multiple threads in a single node are combined locally depending on the shared memory technique chosen by the application developer. After local combination, the results produced by all nodes in a cluster are combined again to form the final result, which is the global combination phase. The global combination phase can be achieved by a simple all-to-one reduce algorithm. If the size of the reduction object is large, both local and global combination phases perform a parallel merge to speed up the process. The local combination and the communication involved in the global combination phase are handled internally by the middleware and is transparent to the application programmer.

Fig. 3 further illustrates the distinction in the processing structure enabled by FREERIDE and map-reduce. The function Reduce is an associative and commutative function. Thus, the iterations of the for-each loop can be performed in any order. The data-structure RObj is referred to as the reduction object.

In the previous section, we had described the complexity of parallelizing wavelet transformation using map-reduce. We can see that that the FREERIDE interface can help simplify the processing. This is because this interface will allow map operations to directly update a reduction object, and not require that (key, value) pairs be sorted.

4. PARALLELIZING WAVELET TRANSFORM ALGORITHM WITH FREERIDE

This section describes the parallelization of the wavelet transformation algorithm using FREERIDE’s reduction-based API.

The main idea is as follows. Let the time-series be of length $T$, and let there be $N$ nodes, so that each node has $T/N$ consecutive values of the input. It turns out that each node is capable of producing $T/N$ elements of the output by processing values locally, i.e., without involving any communication. In all, $T - N$ of the $T$ values can be produced without any communication, whereas the remaining $N$ values involve interprocess communication. Thus, a reduction object of size $N$ can be allocated and initialized on each node, and each node can update it with its own contributions towards the final values. Subsequently, a global reduction can be performed to obtain these final $N$ values.

One issue with this approach is that the output series elements are not going to be consecutive. To allow construction of the final transformed series, we compute and store the final output index with each value.

We explain this approach with the help of an example. Let us consider the following input sequence: $a_1, a_2, \ldots, a_{32}$ for which the wavelet transform needs to be calculated. Let us assume the following: Let the number of nodes be 2, and the let there be 4 threads per node. This implies that node 0 has values $a_1, a_5, a_9, a_{13}$ and node 1 has the values $a_{17}, a_{21}, a_{25}, a_{29}$. Each thread will be responsible for 4 values. In this case, the size of the reduction object during a local accumulation phase, i.e., for finalization of values across the threads on each node, is $\#Threads$, or 4.

The parallel algorithm can proceed with the following steps:

1. Thread 0 can compute the values $a_1 - a_2, a_5 - a_4, a_9 + a_2 - a_3 - a_4$. It then calculates $a_1 + a_2 + a_3 + a_4$ and updates the first value of its local copy of the reduction object with this value. It also sets the remaining values in its copy of the reduction object to 0. Thread 1 simultaneously calculates $a_5 - a_6, a_9 - a_8$, and $a_9 + a_6 - a_7 - a_8$ locally. Then, it updates the second value of its copy of the reduction object with the value $a_5 + a_6 + a_7 + a_8$, and sets the remaining values to zero. Threads 2 and 3 compute their local values similarly and update their reduction objects with $a_9 + a_{10} + a_{11} + a_{12}$ and $a_{13} + a_{14} + a_{15} + a_{16}$, respectively. Thus, the reduction object at the Node 0, which is of length 4, has the values $a_1 + a_2 + a_3 + a_4, a_5 + a_6 + a_7 + a_8, a_9 + a_{10} + a_{11} + a_{12}$ and $a_{13} + a_{14} + a_{15} + a_{16}$.

2. All the threads call an accumulate function whereby the reduction objects of all the threads are accumulated, and one copy of the reduction object is created on each node. Here, the master thread accesses the reduction object and runs the same algorithm for the data on the reduction object. This step produces 3 local final values and one value, which is the sum of all the 16 elements of the reduction object. This value now needs to be communicated among the nodes.

3. We now need to accumulate the values across nodes to deter-
mine the last \#Nodes values of the output series. For this, a global communication step is performed where each node updates its corresponding part on the global array and all the values are combined to form the global reduction object.

We now explain the details of these steps in the following section. The initial implementation is also shown in Figure 4. As we have explained earlier, our implementation is driven by the needs of wavelet transforms on fMRI data. Thus, we consider a 3-dimensional space (volume), where a time-series is computed for each voxel.

```plaintext
Figure 4: Hybrid Parallel Wavelet Transform Implementation
```

### 4.1 Shared Memory Parallelization

Consider parallelization across threads within a single node. The input values are shared between the threads within a node. The access of each thread is restricted to its corresponding \( N_t = \#InputElements / \#Threads \) elements. As we have explained above, each thread can calculate the \( N_t - 1 \) values of the output independently. Each thread also contributes the sum of all the elements to one element of the reduction object. This is done by the accumulate call in Figure 4.

At this stage, we have the \#Threads partial contributions of each thread towards the last \#Threads values. The next step in shared memory parallelization is to extract these \#Threads partial values from the appropriate positions in the reduction object and run the same algorithm on this data. This will produce the last \#Threads values of the output series.

### 4.2 Distributed Memory Parallelization

Now, consider the case when we use one core on each node, but have several nodes. The parallelization is quite similar to the shared memory parallelization. The input data is split among the nodes and each node does the same work that would be done by each of the threads in the shared memory case. Each node calculates the final values that it can calculate locally, and then updates the reduction object with the calculated values in its corresponding position on the reduction object. This is followed by a global combination step, where the reduction objects of all the nodes are accumulated. This is shown as the global_accumulate call in Figure 4.

Since each node calculates \( N_p - 1 \) values, where \( N_p = \#InputElements / \#Nodes \), there are still \#Nodes values that need to be calculated. For this, the sum of all values of each node is retrieved from the reduction object. The wavelet transform is again run on these retrieved values in the master node to generate the last \#Nodes output values.

### 4.3 Hybrid Parallelization and an Optimization

In a hybrid environment, the input data is distributed among the different nodes and multiple threads within each node share the data.

The version of FREERIDE initially available for this study only allowed programmers to declare a single reduction object, for both shared memory and distributed memory parallelization. Thus, the size of the reduction object for our initial implementation (Figure 4) was \#Nodes \times \#Threads. This denotes the number of values in the output that need to be accumulated across nodes or threads. After processing on each thread, each thread updates a part of the reduction object. It turns out that threads within the first node update only the first \#Threads values, the threads within the second node only update the next \#Threads values, and so on. However,
because of the current API and implementation, the entire reduction object is first accumulated on each node, and then communicated across threads. To help reduce communication, we performed an optimization, which required some changes to the FREERIDE API. We separated the reduction object that needs to be communicated across threads.

Recall that we are performing wavelet transform over time-series in a 3D volume. Thus, the total size of the reduction object can get very large. Moreover, an unnecessarily large fraction of the processing is performed sequentially on each thread. To help reduce the communication, we performed an optimization, which required some changes to the FREERIDE API. We separated the reduction object that accumulates values across threads within a node, and the reduction object that needs to be communicated across threads. Thus, for each time series, only the partial values for each time-series are final values and the number of time steps in the output values for the final output values and the #Nodes partial values for each time-series. This is used to calculate the last #Nodes values for the wavelet transform for each of the points in the 3D voxel.

Figure 5 shows how the hybrid parallelization technique with optimization works.

4.4 Generating Index For Output Values

It was shown that an input value at index i will contribute to values at indices i/2, i/4 + N/2, i/8 + N/2 + N/4, \ldots, N − 1, and N. This, together with the node identified can be used to generate the index in the output array where the current value being computed should go. This process is shown in Figure 6.

5. EXPERIMENTAL RESULTS

In this section, we evaluate the performance of our parallel implementation of the wavelet transformation algorithm, focusing on parallel scalability on both distributed and shared memory settings. We performed our experiments on a cluster of multi-core machines, where each node comprises a dual Intel Xeon CPU E5345, with a quad-core CPU. Thus, each node has 8 cores. Each core has a clock frequency of 2.33 GHz and each node has 6 GB main memory. Eight such nodes in the cluster are connected by Infiniband.

We experimented with datasets of different sizes and characteristics by varying two parameters of the dataset, the size of each dimension of the spatial cube p and the number of time-steps in the time-series (s). Thus, the dataset size is \( p^3 \times s \). We generated five datasets with the following dimensions.

1. \( p = 10; s = 262144 \) (DS1)
2. \( p = 32; s = 2048 \) (DS2)
3. \( p = 32; s = 4096 \) (DS3)
4. \( p = 32; \ s = 8192 \) (DS4)
5. \( p = 39; \ s = 8192 \) (DS5)

5.1 Performance of Shared and Distributed Memory Parallelization

In this subsection, we evaluate the speedups achieved from shared memory parallelization mode (1 multi-core node) and distributed memory mode (multiple nodes with 1-core each), separately. Our goal was to understand the scalability achieved with each mode of parallelization. We report results from the datasets DS4 and DS5 in this subsection.

**Results from Shared Memory Mode:** Figures 7 and 8 shows the scalability on datasets DS4 and DS5, respectively. The results are very similar, with speedups of 1.98/1.99 for 2 threads, 3.37/3.40 for 4 threads and 5.50/5.47 for 8 threads. While the application is scaling with increasing number of threads, it is not linear. This is because of the need for combining \( O(p^3) \) values, and also because of limited memory bandwidth available on each node, which slows down the processing when 4 or 8 cores are used.

**Results from Distributed Memory Mode:** Figures 9 and 10 show scalability of the parallel implementation in distributed memory platform for DS4 and DS5 datasets, respectively. In this case, we
are using only 1-core from each node. The overall scalability now is very close to linear, and better than the shared memory case, despite the need for communicating $O(p^3)$ values from each node, and combining them. This further confirms that the shared memory parallelization is limited because of the limited memory bandwidth on each node.

### 5.2 Scalability of Hybrid Parallelization

The results from hybrid distributed-shared memory platform for the datasets DS1, DS2, and DS3 are shown in Figures 11, 12, and 13, respectively. In this set of experiments, we varied both the number of nodes as well as the number of cores used within each node. The dataset DS1 is very distinct from DS2 and DS3, since it involves much fewer spatial coordinates, but has a longer time-series. The size of the reduction object, and therefore the volume of communication per node or thread is $O(p^3)$. Since the time for communication increases with increase in number of threads, there is a slight increase in time taken for 64 threads as opposed to 32 threads (8 nodes running 4 threads each).

While presenting results from the remaining two datasets, i.e., DS4 and DS5, we also evaluate the impact of the optimization we performed for reducing the size of the reduction object.

The performance of hybrid parallelization with and without optimization is shown in Figures 14 and 15 for DS4, and for DS5, they are shown in Figures 16 and 17. Without optimization, the performance increases on multiple nodes up to two threads. Beyond two threads, the increased communication with a large reduction object (before optimization) limits the increase in performance for 4 and 8 threads. Without optimization, the speedup achieved on 64 cores is around 8 for both DS4 and DS5. With optimization, the speedup increases to 25.08 for DS4 and 24.45 for DS5. The overall speedup of optimized versions for DS4 and DS5 are lesser than that of DS1. This is because both DS4 and DS5 have a bigger reduction object when compared to DS1, their speedups are not as high as that of DS1.

### 6. CONCLUSIONS
We have reported on parallelizing the Wavelet Transform algorithm using a reduction framework suitable for cloud environments. We have shown how the Wavelet Transform algorithm can be expressed as a generalized reduction. Using the same high-level, reduction based API, we are able to achieve both shared memory and distributed memory parallelization. We achieve a speedup of up to 42 with 8 nodes running 8 threads each (i.e. 64 threads).

## 7. REFERENCES


